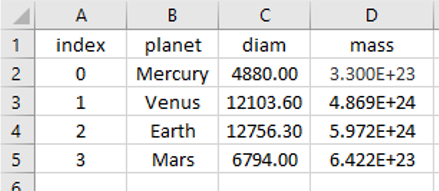
**The Power of ArrayLists**

The structural organization of an array and an **ArrayList** closely resemble one another. These kinds of data structures exist all around if you know what to look for. Bingo cards, rows and seats in a movie theater or an airplane, parking lot spaces, post office boxes, a waiting list at a restaurant, storage cabinets, a marching band, trees in an orange grove, etc. Any place that you see an orderly arrangement of objects or data, you are looking at a concrete example of a data structure. When you spot them, decide whether they would be better represented by an array or an **ArrayList**. The more practical computer science examples you recognize the better!

When an **ArrayList** object is instantiated, think of it as a row in a spreadsheet and each column as an attribute. Dot notation can also be used to represent this organization:

planet(0)  
.name = Mercury  
.diam = 4880  
.mass = 3.300E+23

planet(1)  
.name = Venus  
.diam = 12103.60  
.mass = 4.869E+24

In order to get objects out of an ArrayList object, use the **get()** method as follows.

PlanetV11 dataRecord = planet.get(0);

This assigns the **Planet** object at index position 0 to **dataRecord**, an **object** of type **Planet**. Now the attributes of this record can be handled using any mutator, accessor, getter, and setter methods of the Shapes class. For example:

int diameter = dataRecord.getdiam();

### Part 1

The next examples in the eIMACS labs require some close study. Pay particular attention to the use of the ArrayList in the Animal class; this is the key to handling **ArrayList** constructors with multiple parameters.

Earlier we discussed the use of for-each loops in iterating over arrays. We [remarked](https://www.eimacs.com/eimacs/mainpage?epid=E2375529250&cid=162149#IterableCollections) that there were other types of objects — "iterable" collections — over which a for-each loop may iterate. The ArrayList is one such iterable collection, as the following code demonstrates:

    ArrayList<String> a = new ArrayList<String>();   
    a.add( "fred" );   
    a.add( "eric" );   
    a.add( "jane" );   
    a.add( "emma" );   
    
    for ( String t : a )   
      System.out.println( t );

[Earlier](https://www.eimacs.com/eimacs/mainpage?epid=E2085043452&cid=162149#BadTypeParams) we remarked that the keywords int, double, and boolean are *not* permissible ArrayList type parameters — that is, we are not allowed to use them as replacements for the formal type parameter E in the expression ArrayList<E>. These are three of the Java keywords that represent so-called *primitive data types*, and ArrayLists are not allowed to contain elements of any primitive data type.

Java requires that each element of an ArrayList shall be of a *reference data type* (that's the technical name for a non-primitive data type). The principal difference between elements of a primitive data type and those of a reference data type lies in how their values are accessed. The value of an object of a primitive data type is accessed directly when we use the name of the object. On the other hand, the value of an object of a reference data type is accessed indirectly via a pointer (or reference) that indicates where the actual value is located. Note, however, that you are *not* expected to understand such technical matters for the Advanced Placement examination.

This state of affairs is unfortunate, because it would be really useful if we could store ints and doubles (in particular) in collections that can grow and shrink just as ArrayLists can. To get around this problem, Java provides a built-in reference data type corresponding to each primitive data type. These special reference data types are known as *wrapper classes* because they "wrap" primitive data type values in the additional layer of complexity possessed by reference data type objects. Each wrapper class must be used in place of the corresponding primitive data type when creating ArrayLists. As far as the three primitive data types we have just mentioned are concerned, the wrapper classes are as follows:

|  |  |
| --- | --- |
| *Primitive data type* | *Wrapper Class* |
| int | Integer |
| double | Double |
| boolean | Boolean |

With the help of these wrapper classes, we operate as follows: If we want to store some ints in an ArrayList, we declare the ArrayList as an

ArrayList<Integer>,

using the wrapper class that corresponds to the int primitive data type. We then add ints to the ArrayList without concerning ourselves about the fact that they are ints rather than Integers — Java automatically performs the necessary conversion. In the other direction, we can retrieve ints from an ArrayList of Integers using the get method. Whenever the result of such a retrieval needs to be an int rather than an Integer, Java automatically takes care of the necessary conversion. The following code, for example, uses an ArrayList to manipulate ints. First, an array of ints is copied into an ArrayList. Then all the 5s are removed from the ArrayList and the remaining elements are printed:

    ArrayList<Integer> a = new ArrayList<Integer>();   
  
    int[] someints = { 5, 2, 5, 7, 5, 8, 5, 2 };   
  
    // store elements of someints into the ArrayList   
    for ( int y : someints )   
      a.add( y );   
  
    // remove all the 5's   
    int q = 0;   
    while ( q < a.size() )   
    {   
       if ( a.get( q ) == 5 )   
         a.remove( q );   
       else   
         q++;   
    }   
  
    // print the result   
    for ( int p : a )   
      System.out.println( p );

The automatic conversions that occur during the execution of this code are as follows:

* Each addition of an int to the ArrayList in the first for-each loop involves the automatic conversion of an int to an Integer.
* Each comparison of 5 with the result of getting an element from the ArrayList involves the automatic conversion of that result from an Integer to an int.
* In the final for-each loop, because the variable p is declared as an int, each successive assignment of an element of the ArrayList to p involves the automatic conversion of an Integer to an int.

When working with ArrayLists, there is nasty little "tiger trap" that must be avoided at all costs: for-each loops and operations that entail changing the size of an ArrayList do not mix. Consider the following example, in which the intention is to create an ArrayList containing the first 20 odd numbers and then remove all those that are multiples of 7.

ArrayList<Integer> a = new ArrayList<Integer>( 20 );

for ( int k = 0; k < 20; k++ )

a.add( 2 \* k + 1 );

for ( int i : a )

if ( i % 7 == 0 )

a.remove( i );

System.out.println( a );

When you press the **Run** button you will find that an error is reported that starts like this:

Exception in thread "main" java.util.ConcurrentModificationException

This happens because for-each loops work their way down an ArrayList using a behind-the-scenes structure called an *iterator*. This keeps track of where the loop is in the ArrayList by using a technique that relies on knowing how long the ArrayList is. If you do anything to change the size of the ArrayList (by removing or adding items or by clearing the ArrayList, for example), you mess up the operation of the iterator, which then throws the ConcurrentModificationException as if to tell you: "Don't change the size of this ArrayList while I'm iterating my way along it!"

There are a number of ways to overcome this problem. One is to use a regular for loop, working from the end of the ArrayList toward the start, like this:

for ( int i = a.size() - 1; i >= 0; i-- )   
    if ( a.get( i ) % 7 == 0 )   
        a.remove( i );

Copy and paste this code to replace the for-each loop in the above code area and press the **Run** button. You'll find that the list is printed of all the odd numbers up through 39 except for 7, 21, and 35.

In the interests of full disclosure, we should let you know that there is much more going on here than meets the eye. In the original for-each loop, because the loop variable i is declared as an int, each Integer in the ArrayList a is auto-unboxed one at a time and its int value is assigned to i. Then in the call to the remove method we are using a version of that method that has a different signature than the one that's in the AP Java subset. The input to the AP Java subset version of remove specifies the *index* of the item to be removed. But there is also a version of remove that accepts an object of the same type as the items in the ArrayList. This causes the item to be removed from the ArrayList that has the smallest possible index and that is equal to the input object. Of course, in this case, since i is an int, some auto-boxing has to occur to transform it into an Integer for the remove method to remove.

In the regular for-loop, the loop variable i is clearly intended to serve as an index. This is evident because the divisibility test requires the use of the get method to obtain the relevant item from the ArrayList, and that item is then auto-unboxed to obtain an int to which the modulus operation can be applied. In this loop, the version of the remove method that is being used is the one that is in the AP Java subset.

### Part 2

Although the principle is the same, **ArrayLists** with multiple attributes are a little more complicated because you first must assign each record to a new object. Once this is done, the support methods of the implementation class can be used. A final version of the **Planet** class is used to demonstrate the use of **ArrayLists** in the following demo program.

* Create a new project called 07.02 ArrayLists in the Mod07 Lessons folder.
* Download the [PlanetV11.java](https://lti.flvsgl.com/flvs-cat-content/8cpcf5itst2jqkc9bi5sovl2mb/flvs-cat-session/apcomputersciencea_v20/module07/lesson02/docs/07_02b/planetv11.java) and the [PlanetTesterV11.java](https://lti.flvsgl.com/flvs-cat-content/8cpcf5itst2jqkc9bi5sovl2mb/flvs-cat-session/apcomputersciencea_v20/module07/lesson02/docs/07_02b/planettesterv11.java) classes to the newly-created project.
* Open the [07.02 Virtual Lecture Notes](https://lti.flvsgl.com/flvs-cat-content/8cpcf5itst2jqkc9bi5sovl2mb/flvs-cat-session/apcomputersciencea_v20/module07/lesson02/pop/07_02b/07_02b_pop1.htm).
* Very carefully read the document and make the recommended modifications to the demo program.